AFRL-RI-RS-TR-2014-249

CODE PULSE: SOFTWARE ASSURANCE (SWA) VISUAL ANALYTICS FOR DYNAMIC ANALYSIS OF CODE

APPLIED VISIONS

SEPTEMBER 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2014-249 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
SEPTEMBER 2014	FINAL TECHNICAL REPORT	AUG 2012 – MAY 2014

4. TITLE AND SUBTITLE

CODE PULSE: SOFTWARE ASSURANCE (SWA) VISUAL ANALYTICS FOR DYNAMIC ANALYSIS OF CODE

5a. CONTRACT NUMBER FA8750-12-C-0219

5b. GRANT NUMBER N/A

5c. PROGRAM ELEMENT NUMBER Other

6. AUTHOR(S)

Hassan Radwan, Dylan Halperin, Robert Ferris, Ken Prole

5d. PROJECT NUMBER DHS2

5e. TASK NUMBER AV

5f. WORK UNIT NUMBER IS

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Applied Visions, Inc.
Secure Decisions Division
6 Bayview Avenue
Northport, NY 11768

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2014-249

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2014-4339
Date Cleared: 12 Sep 14

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This is the final report for Code Pulse, a BAA project by Secure Decisions in the software assurance space. The original intent was to enhance static application security testing by introducing execution profiles of target applications to the vulnerability analysis process. However, as the concept was introduced to evaluators via prototypes, the feedback steered the project towards implementing a real-time code coverage tool for penetration testing activities. Code Pulse is an open source tool that provides a real-time white box perspective of coverage activity for penetration testers as they conduct their testing activities. This report details the activity for this project along with describing the evolution of the concept from original vision, to end state.

15. SUBJECT TERMS
Application security, penetration testing, black box, white box, software assurance, dynamic analysis, DAST, interactive application security testing, IAST, tracing, runtime, real-time, code coverage, visualization, static analysis, SAST, cyber security.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**WILMAR SIFRE**
U	U	U	UU	43	**19b. TELEPHONE NUMBER** *(Include area code)* **315-330-2075**

Contents

List of Figures

List of Tables

1 EXECUTIVE SUMMARY

In this effort funded by the Cyber Security Division of Department of Homeland Security (DHS) Science &Technology Directorate, Secure Decisions researched and evolved a concept to integrate runtime observations of target software in application security assessment processes. The original intent of this project was to enhance static application security testing by introducing execution profiles of target applications to the vulnerability analysis process. However, as the concept was introduced to evaluators via prototypes, the feedback steered the project towards implementing a real-time code coverage tool for penetration testing activities. Code Pulse is an open source tool that provides a real-time white box perspective of coverage activity for penetration testers as they conduct their testing activities.

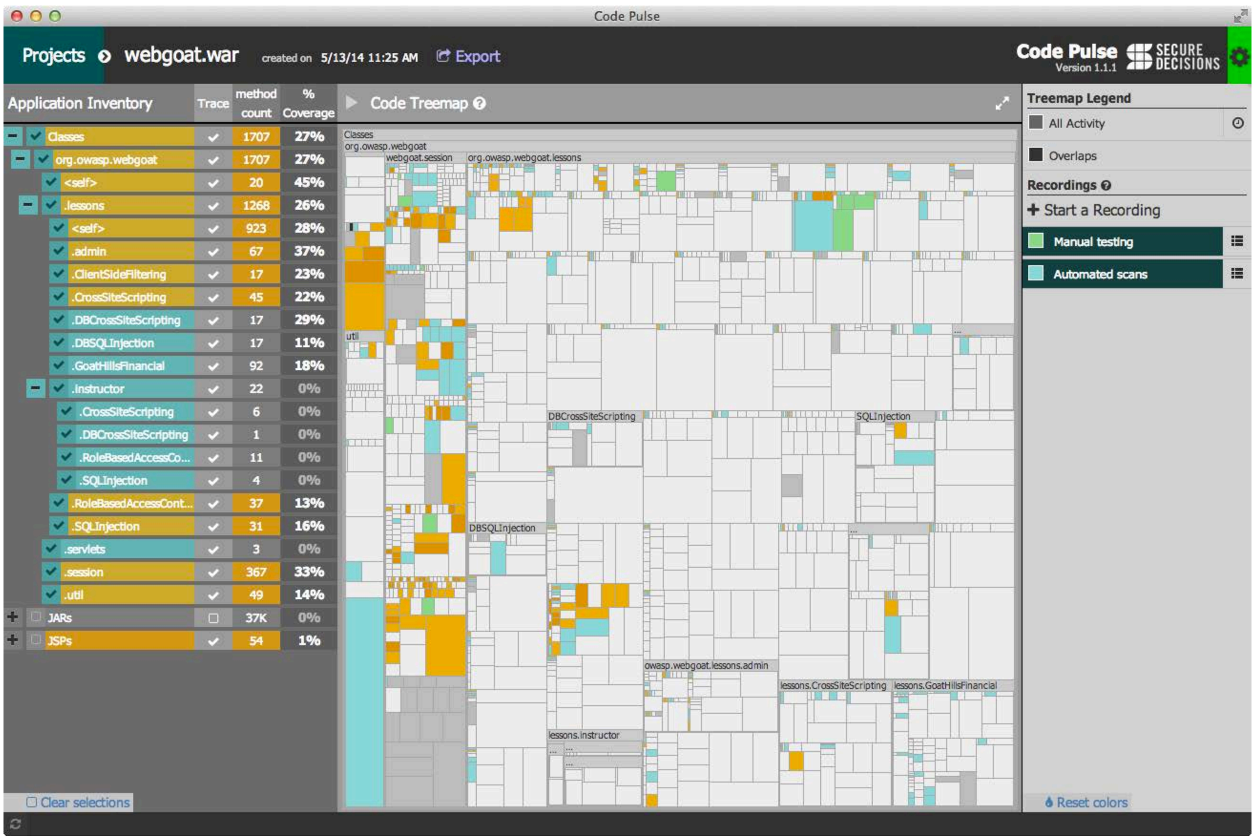

Figure 1. Code Pulse

The Code Pulse project effort revolved around three key milestones: an alpha prototype to demonstrate to early evaluators and validate the concept; a beta prototype to fine tune the utility and usability of the tool; and finally the release of the first stable version as a TRL6 open source tool part of the vibrant Open Web Application Security Project (OWASP) project inventory. This report details the problem areas addressed by Code Pulse, the adopted methodologies to steer the project, the results of the varied project activity, and finally the conclusions we draw at the end of the project's period of performance.

2 INTRODUCTION

Application security tools have up until recently focused on either a white box static analysis of target codebases, or alternatively a black box dynamic analysis for penetration testing. More recently we've seen a new breed of security applications aiming to introduce a white box *dynamic* perspective to the analysis process. Some of these tools classify themselves as Interactive Application Security Testing (IAST) tools. Although Code Pulse went through an evolution of its focus area during its period of performance, two constant themes have been the belief that runtime observations of target codebases provide a wealth of information that will improve the analysis processes of application security testing, and the recognition that it is best to present this information in a visual analytics manner to improve the understandability and communication effectiveness of the relevant data.

In this section we'll describe the problem areas targeted by Code Pulse and the evolution of our thinking during the period of performance as we talked with evaluators and gained a better understanding of their needs.

2.1 Code Pulse Evolution

The originally proposed concept of Code Pulse was to introduce a new technique in software assurance analysis by merging application runtime observations with static source code analysis to enhance static analysis workflows. This relies on harvesting execution profiles from the runtime execution of the target codebases. Key factors are collected and stored from the execution profile, such as when each part of the code is used in the application and how frequently it is used. These in turn are correlated with the results of static analysis tools to identify the presence of a weakness during the application execution, as well as how many times the potentially vulnerable code was executed. We coined the term *Dynamic Enhanced Static Analysis* (DESA) to describe the approach and you'll see it referred to in various places in this report as a short-hand for the concept. The motivations for this approach are described in section 2.2.

Secure Decisions both advocates and follows an evaluation-based development approach. This relies on getting user feedback as early as possible to course correct earlier rather than later when it's often too hard or even impossible. Our first demonstration milestone for Code Pulse was an alpha prototype which we built to target the use cases we identified for the DESA approach. The Code Pulse prototype was presented to several evaluators and ended up getting a lukewarm reaction. The listing of evaluators and the results discussion is detailed in section 4.4.2. This gave us reason to pause and reflect on the feedback, ultimately resulting in a direction shift for Code Pulse.

With the technical capabilities we'd built up for the alpha prototype, and the areas where we got positive (and even enthusiastic in some cases) feedback, we decided, with our Program Manager's blessing, that the best course of action was to produce a tool that focused on code coverage for penetration testing activities. The motivations for that are described in section 2.3 and the resulting tool is presented in section 4.3.5.

2.2 Static Analysis Workflow Problem Area

The typical static analysis process involves a number of user personas, but for the purposes of the following description we'll focus on just two: the security analyst assessing the security posture of a target codebase, and the developer tasked with fixing vulnerabilities as they are reported. Most security analysts will rely on one or more Static Application Security Testing (SAST) tools to identify potential vulnerabilities in the codebase. These results will typically number in the thousands of findings and are triaged by the security analyst to reduce them to a manageable short list of prioritized issues. Developers are at that point tasked with addressing the prioritized list of vulnerabilities. This cycle repeats itself for every security assessment dependent on the motivation and practices of the concerned parties. Although this is a simplified description of the workflow, there are two inefficiencies that would benefit greatly from an improved process.

The first inefficiency is the broad net cast by the SAST tools which will assess all areas of the codebase. However, if you consider the realities of a typical codebase, you'll find a lot of extraneous code for things such as unit testing, database maintenance scripts, application packaging scripts, and the like. While this code is relevant for the development and deployment teams, it is irrelevant to the attack surface which ultimately is the desired area of focus for security analysis assessments. Therefore, focusing on only the relevant weakness, those that are part of the attack surface, would significantly boost the efficiency of both the triage and remediation activity for security assessments.

The second identified inefficiency in the SAST workflow relates to the timing of when developers are asked to remediate a vulnerability. In our current software development realities, security assessments are few and far between. This translates into remediation requests to the development team weeks or months after they've last looked at the source code. A number of studies have been conducted on both the short and long term memories of developers and the results have shown that there is only so much context that developers can keep in memory. Therefore, with the typical assessment frequencies, by the time developers are asked to remediate a weakness they've most likely forgotten the context that would enable them to effectively and efficiently dispatch the vulnerability.

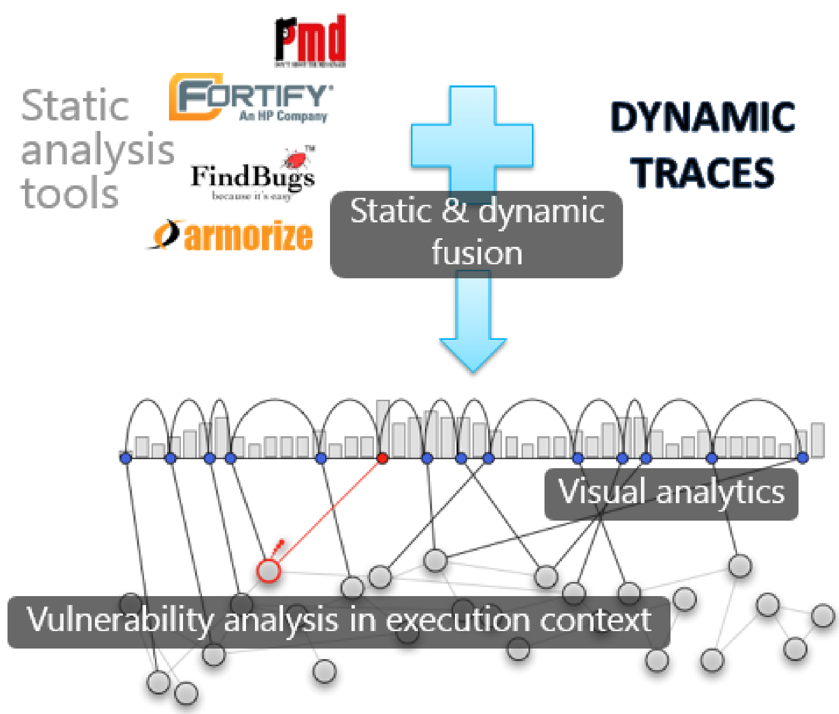

Figure 2. Code Pulse approach to fusing SAST results with dynamic traces

The Code Pulse approach to this solution, represented in Figure 2, was the target for our alpha prototype. The concept was to fuse the results of static analysis tools with dynamic traces and represent the correlated data using visualizations. This served two key use cases: prioritization of weaknesses by security analysts/auditors; and remediation activities by developers. Identifying which code was executed and correlating that with the statically observed vulnerabilities would help security analysts identify which weaknesses are part of the attack surface. Visually representing the call graph and execution timeline to developers would help them quickly remember the context of the remediation they're addressing as well as effectively understand what implications their changes would have.

2.3 Penetration Testing Code Coverage Problem Area

Penetration testing, also known as black box testing, has proven to be a highly effective approach to identifying security issues in target applications. Penetration testers leverage a number of manual techniques and Dynamic Application Security Testing (DAST) tools to probe the target applications for any vulnerabilities that can be exploited. As exploits are discovered they are catalogued and presented to the application maintainers for fixing. While penetration testing is a useful and increasingly used process, there are certain issues in the process that need addressing:

- **Coverage gaps –** by definition, penetration testing is typically a purely black box perspective which makes it almost impossible to ascertain the attack surface coverage gaps. How much or little of the attack surface was assessed is almost impossible to ascertain without other means of understanding the application scope and how that correlates to the testing activity.
- **Unclear coverage overlaps –** understanding the coverage overlaps between penetration testing techniques – manual testing and/or multiple DAST tools – is very difficult. Each represent the testing activity differently making it a considerable challenge to correlate activity across tools.
- **DAST tool tuning –** DAST tools are tricky to configure due to the complex variations in the target applications. Knowing when they're misconfigured is critical to ensuring that adequate coverage of the target applications is achieved.
- **Coverage data communication –** communicating coverage data is a significant challenge due to the lack of coverage insight from the black box perspective. Yet despite the challenge, coverage data and the ability to communicate it is critical since it is ultimately the true measure of whether an application has been adequately tested or not.

The Code Pulse approach to these set of challenges is to leverage dynamic tracing to represent the code coverage in real-time to penetration testers while they are conducting their tests. While penetration testers are conducting their tests on the target applications, Code Pulse traces the application to expose its codebase logical hierarchy and extract the coverage data. This is presented to the user in real-time visually providing them with a white box perspective into the application coverage while they conduct their black box testing. This concept, adopted by Code Pulse after the alpha prototype milestone, is represented in Figure 3.

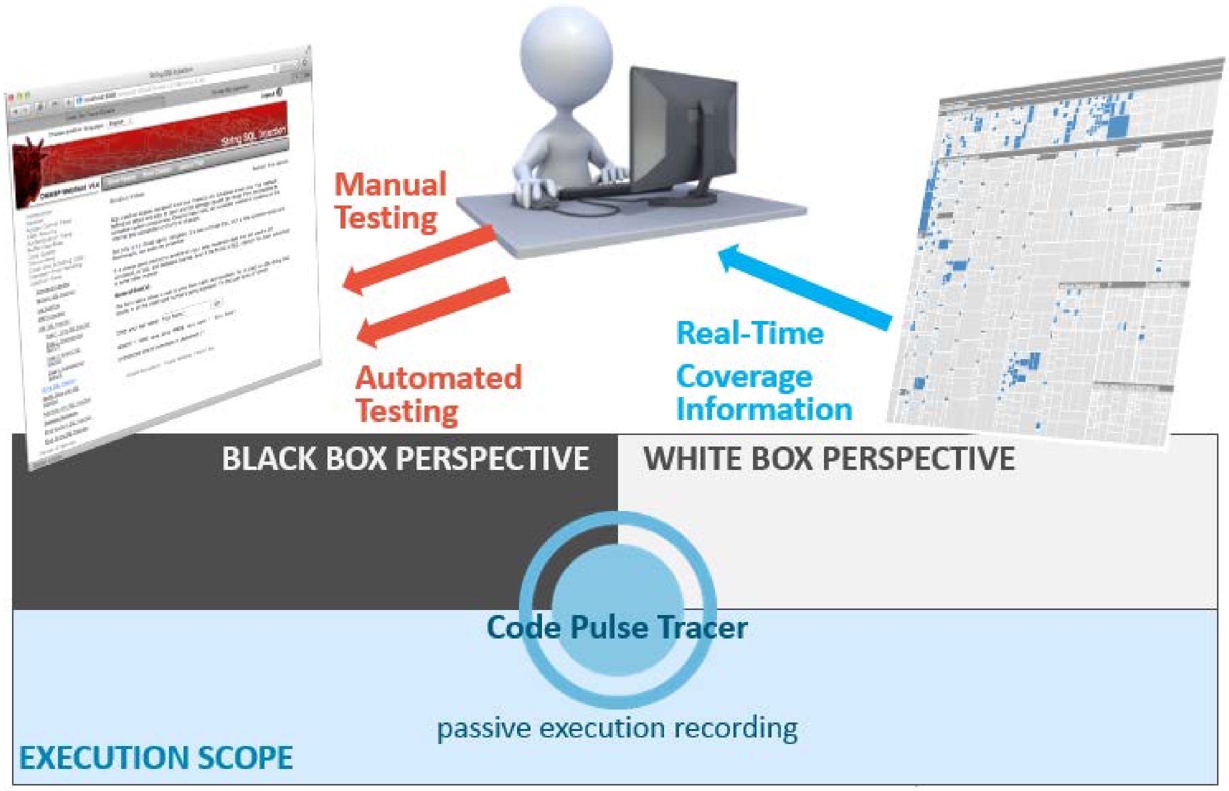

Figure 3. Code Pulse approach to penetration testing code coverage

This approach has a number of benefits:

- **Visually identifies coverage gaps –** coverage gaps are identified visually and can be understood at-a-glance
- **Compares coverage across testing tools –** coverage data can be separated to identify the activity by tool or technique as we as identify overlaps between the varied activity
- **Communicates coverage activity –** the coverage data is automatically processed and displayed to the users making it significantly easier to communicate that information with the other stakeholders
- **Eases the DAST tuning process –** visually seeing the coverage data quickly helps uncover DAST tool misconfigurations for quick and effective tuning

3 METHODS, ASSUMPTIONS, AND PROCEDURES

The Code Pulse project was managed using a modern iterative software/systems engineering approach. The cornerstones of our approach were research, design, development, testing, and evaluation. These were done iteratively with constant assessment and planning of the transition potential and strategies throughout the process. Two distinct prototypes were generated throughout the period of performance and presented to evaluators. The received feedback proved to be critical to the project, helping shape the development and transition directions as the project evolved.

Our effort on Code Pulse was split across six tasks, which although considered separate from a focus area perspective, where often conducted simultaneously and iteratively. The methodologies, assumptions, and procedures for each of the Code Pulse thrust areas are detailed in the remainder of this section.

3.1 Task 1 – Investigate Dynamic Analysis Tools

A key component of the Code Pulse approach is to trace applications at runtime to observe and record an execution profile. Therefore, the selected technology used to provide the tracing capability is central to the functionality of Code Pulse. To mitigate risks associated with the technology selection, the first task for Code Pulse was to identify the project's data requirements, research the available dynamic tools/libraries, identify the limitations of each of those tools/libraries, and finally make a tool/library selection to base the project's development around. The details of this activity are presented in section 4.1.

3.2 Task 2 – Design Prototype

Although we're strong proponents of leveraging agile and iterative methodologies for software development, identifying the use cases and designing a user experience around that is a critical first step in any software project. Therefore we started our Code Pulse design efforts by identifying the key project use cases, listed in more detail in section 4.2. With a solid understanding of the use cases and user personas involved, we designed the user experience that would satisfy both the use cases and our requirements. This came in the form of low fidelity mockups and wireframes, described in detail in sections 4.2.4 and 0.

What followed thereafter was a process of iterative design that went hand-in-hand with the development, testing, and evaluation activity on the project as Code Pulse took form throughout the period of performance.

3.3 Task 3 – Develop Prototype

Implementing the vision as expressed via the use cases and user experience design was the primary objective of this task. To achieve that goal, three key milestones were set as the development targets for the project. The first, was an alpha prototype serving as the first demonstrable version of Code Pulse to be used in evaluation sessions for feedback. The second milestone was a beta prototype encompassing a more complete and refined version of Code Pulse based on the feedback gathered from prior feedback. The third and final milestone was the first stable release, version 1.0, of Code Pulse to be used by security professionals in their software assurance activities.

Identifying the engineering direction was the first focus of this task. This involved coming up with the system design and architecture as well as identifying the technology stack that would best satisfy the requirements and designs. Both of these are described in detail in sections 4.3.1 and 4.3.2.

We've long been adopters of agile methodologies in our software development practices and followed a Scrum-like process for Code Pulse development. This typically involved focusing our development activity around 2-3 week iterative Sprints. Conducting our development using these periodic checkpoints gave us the opportunity for introspection to continuously evolve our engineering and user experience designs during the early stages of the project, and performance and usability fine tuning in the latter stages.

3.4 Task 4 – Test, Demonstrate, and Evaluate

This task served as an umbrella for three key efforts: software testing, demonstrations to potential users and interested parties, and conducting evaluations of the system to serve as a feedback mechanism to steer development efforts. Activity in this task revolved around the alpha, beta, and version 1.0 milestones described in the previous section. This task was kicked off by drawing up a Test Plan, submitted on 4/30/2013, outlining the strategy for testing the application throughout the period of performance. Using the Test Plan as the blueprint, testing was conducted throughout the development stages, although it was intensified in the run-up to the milestones to adequately ensure the stability and usability of Code Pulse. With both the alpha and beta prototype, a round of demonstrations and evaluations was conducted to assess the perception of the tool's utility, its usability, as well as the user buy-in into the presented concepts. The details of this task's activities are detailed in section 4.4.

3.5 Task 5 – Prepare for Technology Transition

Determining the technology transition path for a research project is essential to ensuring its survivability beyond the period of performance. Perhaps more importantly, it also provides the focus that a clear and defined target brings to the table. Efforts on this task were started by formulating an initial project vision to define the expectations for the final state at the end of the period of performance. The transition potential was further researched and explored which resulted in the first draft of the Technology Transition Plan document, submitted on 6/6/2013, outlining our research and initial project strategy. As Code Pulse crystalized its capabilities through feedback, renewed focus was brought to the technology transition efforts to form an updated plan around the evolving tool realities. Transitions options were further researched along with an in-depth competitive market analysis to assess the tool potential. The final transition targets were selected and expressed along with our research on the topic in our updated Technology Transition Plan document, submitted on 5/30/2013.

3.6 Task 6 – Manage and Document

Staffing, budgeting, scheduling and reporting were the primary activities conducted under this task. Selecting the project team and responsibilities from within the Secure Decisions talent pool was one of the initial activities conducted for this task. Tracking our progress relative to our objectives was a constant activity as we tracked the budget and schedule. Finally, keeping our sponsor appraised of our progress via the required quarterly technical status, monthly financial reports, PI meeting presentations and discussions, and informal conversations with our Program Manager was essential to ensuring the project remained on track to match our sponsor's expectation as the project direction evolved throughout the period of performance.

4 RESULTS AND DISCUSSION

This section outlines the results of the Code Pulse efforts during the period of performance.

4.1 Tracing Tool Research

The end goal of this task was to select a dynamic tracing library that best meets the needs of the project. This library would then be integrated into the developed solution to monitor target applications during runtime and collect the relevant data for weakness correlation.

4.1.1 Evaluation Criteria

To start off this activity, we established the evaluation criteria. These were drawn up based on a variety of requirements ranging from performance considerations, to specific data constraints needed to satisfy the project use cases. The list of the established evaluation criteria is as follows:

- **Maturity** – a mature and well-established tool will have sufficient testing to ensure maximum compatibility.
- **Non-invasive instrumentation** – to minimize impact on the target applications a constraint was established to ensure that any selected tool would leverage a passive instrumentation technique that did not modify the source code or binary. This would make the tool adoption an easier process since no changes would be required by the software development team.
- **Light-weight** – tracing a target application would have an impact on its resource allocation. To minimize resource contention a criterion was established to ensure that any selected tool would be as minimally impactful as possible on the execution of the target applications.
- **Licensing** – to enable the integration of the selected tool within the final Code Pulse solution, the license needed to be non-viral. So tools with a General Public License (GPL) and other similar viral licenses were excluded from consideration.
- **Data constraints –** the following were specific data constraints needed by the Code Pulse uses cases that the tools needed to satisfy (note that the associated timestamp for each of the data points is required):
    - Identify thread creation and destructions
    - Identify method entries and exits with the associated thread identifiers
    - Identify exceptions with the associated thread identifiers and locations in the source
    - Provide a mechanism to filter out source code elements from the tracing

4.1.2 Evaluated Tools

Basing our initial selection on the evaluation criteria we performed a survey to identify the potential candidate tools. The identified tools along with a brief description of each is listed as follows.

- **AspectJ**[1] – an open source library developed for aspect oriented programming. While tracing is not the primary use case of AspectJ, the mechanisms used are very similar and therefore we selected it to test its bytecode instrumentation capabilities.
- **JVMTI** – the Java Virtual Machine Tool Interface is a monitoring framework provided by the Oracle Java Virtual Machine (JVM) to observe various metrics of the Java envi-

ronment. Although it has more to offer, one of JVMTI's key capabilities is monitoring execution traces.

- **BTrace –** an open source project aiming to bring DTrace's renowned utility to Java.
- ***J –**another open source project aiming for low impact java execution tracing.
- **VisualVM –** a powerful profiler maintained by Oracle and bundled with the Java Development Toolkit (JDK). As a profiler, one of its key features is monitoring execution traces. Although VisualVM is a full featured application, we were hoping to leverage its tracing backend.
- **JDI –** the Java Debug Interface (JDI) is the debugging Application Programming Interface (API) for the JVM. Thanks to its deep integration into the runtime of the debugged applications it can be leveraged to extract the execution trace information.
- **BCI –** although Byte-Code Instrumentation (BCI) is a technique rather than the name of a library or tool, it is a technique that relies on an instrumentation API that is part of the Java language. Most modern profilers rely on it and similarly there was potential for us to leverage it to monitor runtime execution.

4.1.3 Evaluation Methodology

A methodology was designed to evaluate the tool candidates. The intent was to test a realistic range of scenarios that the tracing would encounter to ensure a realistic evaluation experience. Evaluations were primarily focused on performance impact, data accuracy, data criteria fulfilment, and robustness. Each of these points is discussed individually in the following paragraphs.

Performance considerations – to evaluate the performance of the dynamic tracing tools, timing tests were conducted on traces of the test applications. A baseline test was conducted first to identify the runtime timing of the application without any added tracing overhead. Subsequent traced tests were timed and compared against the baseline timing to determine the slowdown factor for each of the evaluated tools. All tests were conducted on the same machine under similar load constraints.

Data accuracy – ensuring the data accuracy was a challenge due to the large amounts of data collected during traces. In addition the data order was non-deterministic for the most part due to the multi-threaded nature of most current-day applications. The approach we derived was to compare event counts across all tools and conduct spot checks of the data. Comparing event counts proved to be valuable as it uncovered a number of differences between the tools and helped us further understand their limitations.

Data criteria – evaluating the data criteria was a straight-forward testing process to identify which if any of the criteria the tool did not satisfy.

Robustness – to evaluate the robustness of the candidate tools we established a set of testing applications used to trace against. The set, listed in Table 1, was designed to be broad enough to test the varied scenarios that the tracing solution would encounter whilst still being small enough to be manageable during the evaluation period.

Table 1. The test application set used for the dynamic tool evaluations

Test Application	Description
Simple single threaded	This is a simple single threaded application that we wrote to have a deterministic trace at hand with expected results we can use to ensure data accuracy.
Multi-threaded sorter	Another test application we wrote to test multi-threaded tracing using a sanitized application with a well-defined execution behavior.
Glaz Treemap Demo	A treemap demo from a Java visualization library used internally at Secure Decisions. It relies on Swing and Java2D and was selected to test out Swing-based applications.
Eclipse	The Java Integrated Development Environment (IDE) that needs no introduction. It was selected to test traces of the Eclipse Rich Client Platform (RCP).
JavaFX Scene Builder	A tool used to visually build JavaFX user interfaces. It was selected to test JavaFX-based applications.
WebGoat	An OWASP sponsored project used to teach software engineers how to write secure source code. It's a web-based application and was selected to test tracing of web applications.

4.1.4 Evaluation Results

Of the seven tools, three were dismissed after short evaluations due to significant issues:

- ***J –** Taking a closer look at *J revealed that it makes use of the Java Virtual Machine Profiler Interface (JVMPI), which was deprecated many years ago since Java 5. In addition the project seems to be abandoned since it hasn't seen any activity since 2004. For these reasons further evaluations of *J were abandoned.
- **BTrace –** BTrace initially seemed promising due to a lightweight tracing approach. It is designed as a minimal impact library and leverages modern bytecode instrumentation techniques. However, after evaluating it on the prepared test applications we discovered that it did not work consistently and outright failed to trace for some of them. Taking a closer look at the codebase we also saw that it had not seen any activity in the past 6 months and did not seem to have an active or mature community. As a result we stopped further evaluations of BTrace.
- **VisualVM –** With VisualVM's established track record and large user base, the assumption was that it had a strong potential to be leveraged for the desired dynamic tracing. While VisualVM itself is an application, the intent was to interface with its backend API. However, as we familiarized ourselves with the tool we discovered that VisualVM does not have a formal API. The development team itself dissuaded developers from using the open source codebase as an API since it was not designed as one and was tightly coupled with their specific solution. A source code examination confirmed that and planned testing with this tool was abandoned.

The remaining tools were evaluated closely both programmatically and with the test applications. The evaluations yielded the following insights for each of the tools.

- **AspectJ –** Of the evaluated libraries AspectJ ended up being the fastest. It uses a BCI approach to instrument the desired code and does so at load time to reduce overhead during runtime. In addition it is a very mature library with a large user-base. We tested AspectJ extensively and whilst it traced most test applications successfully, there were certain target applications that AspectJ was unable to trace. After further investigations, including looking at the source code for modifications, it became clear that attempting to modify the AspectJ source to suite our requirements would yield a brittle solution and take considerable effort. Ultimately, we decided to abandon the prototypes we built up around AspectJ and settle on an alternate solution.
- **JVMTI –** To evaluate JVMTI we created a simple framework to interface with its API and monitor target applications on the host JVM. Overall JVMTI worked well tracing the test applications as expected. However, three downsides led us to eventually dismiss JVMTI despite it being a close contender: the first was that it required that native code be written to interface with it which would have complicated the development plans since the target development stack was on the JVM; the second was that JVMTI did not support late attach-mechanisms to monitor target applications *after* they had already started up; and the third reason was that it had a significantly higher performance slowdown on the target applications than techniques relying on BCI for tracing.
- **JDI –** Of the dynamic tracing tools, JDI provided the richest insight into the traced applications behavior. However, it was also the slowest by a factor of 100. This is understandable since the debug use case is different than the tracing one. But with the significant slowdown in the target application's speed we dismissed JDI as a contending tool after evaluating it on some of the test applications.
- **BCI –** Initially we did not fully evaluate the built-in BCI libraries since doing so would have been a significant time commitment outside the scope of the initial evaluations. However, in order to familiarize ourselves with the technique we did build a small layer on top of it and tested it briefly to get a sense of the complexities in developing such a solution. The conclusions we drew that going with a custom BCI solution would offer the largest degree of flexibility, however, it required a non-negligible commitment to create the bytecode injection layer necessary to instrument the target applications with.

Ultimately we realized that creating a custom tracing solution using the Java Virtual Machine (JVM) byte-code instrumentation APIs would be the most effective and flexible long-term approach. The contending solution, AspectJ, was not designed for tracing use cases, and modifying it to support those use cases more readily would have been a significant change to a large and complicated codebase. Whereas leveraging the byte-code instrumentation APIs allowed us to customize the solution for our needs and offered us a familiar codebase that we can maintain going forward to support requirements changes as they invariably arise.

The culmination of this task was to select the custom byte-code instrumentation implementation as the most viable approach for the needs of this project.

Table 2. Tracing numbers with some of the test applications

Application	Base Timing	Traced Timing	Slowdown Factor	Num. of Events	Trace Size
Sorter	1.5s	2.8s	1.9	6,385,006	79.1MB
Glaz Tree Map Demo	10s	237s	24	400,881,532	4.85GB
Startup	1.9s	90s	47		
Search	<1s	55s			
WebGoat		242s		2,543,046	32MB
Startup	0.5s	3s	6		
Homepage	0.67s	0.92s	1.4		
HTTP Basics Lesson	0.024s	0.054s	2.3		

4.2 Design

4.2.1 Dynamic Tracing Use Cases

Tracing target applications is a key activity that will be conducted by the users to collect the relevant runtime data for observation and further analysis. The high-level Dynamic Tracing (DT) use cases identified during our initial design phase are listed in Table 3.

Table 3. Dynamic tracing use cases

Dynamic Tracing (DT) Use Cases	
DT1	Manually record the execution trace of a target application from the initial load
DT2	Manually record the execution trace of an already running target application
DT3	Automatically record the dynamic trace of an application in a headless setup environment
DT4	Configure a dynamic trace to exclude specific packages from the recording

4.2.2 Dynamic Enhanced Static Analysis Use Cases

For the dynamic enhanced static analysis approach, we identified five key high-level use cases. The Code Pulse with Code Dx integration (CPDx) use cases, listed in Table 4, describe Code Pulse specific static source analysis interactions within Code Dx.

Table 4. Dynamic enhanced static analysis use cases

Dynamic enhanced static analysis (DESA) Use Cases	
DESA1	Triage only the weaknesses in an execution context
DESA2	Set all weaknesses in critical code paths to the highest priority
DESA3	Set all high severity weaknesses in the least frequently traversed code to the highest priority
DESA4	For a given weakness, observe the different code paths that lead to it
DESA5	For a specific weakness, observe the timing of its execution during the collected runtime traces

4.2.3 Penetration Testing Code Coverage Use Cases

The following use cases in Table 5 were identified for the penetration testing code coverage approach.

Table 5. Code coverage use cases

Penetration Testing Code Coverage (CC) Use Cases	
CC1	Identify the application inventory
CC2	Identify the coverage gaps of the target codebase
CC3	Identify coverage overlaps across different testing techniques and tools
CC4	Share the coverage data with other stakeholders

4.2.4 User Interface Design

To solidify the user experience design, a series of wireframes and mockups were created and iterated over as we evolved our thoughts on the user interaction and data flow within the application. Although the user interface design was a continuous process throughout the development of the application, design efforts intensified at the start of the development milestones. Initially the process involved designing the entire user interface as it was conceived. Progressively, this became more a process of tweaks and refinements and the designs matured and solidified.

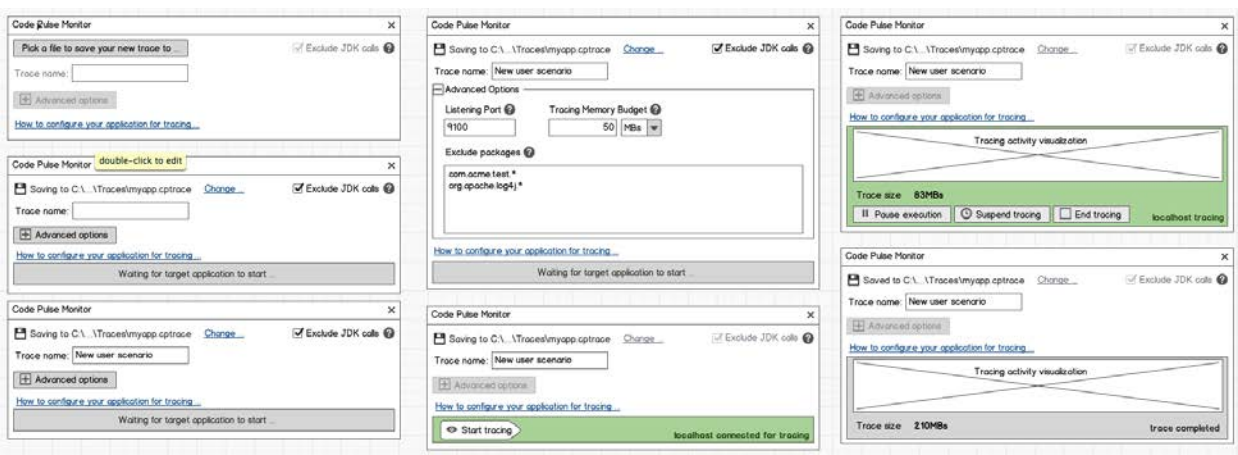

Figure 4. Early wireframe of a standalone interface for the Code Pulse dynamic tracer

Since Code Pulse was intended to be a data intensive application, visualizations were key to the interface design to communicate the data in an effective and actionable manner. Although we eventually settled on leveraging the increasingly familiar treemap visualization to display the codebase coverage data, we explored various visualization techniques for different parts of the interface as we evolved our user experience design. Figure 5 shows a mockup of how different visualizations could be leverage for Code Pulse.

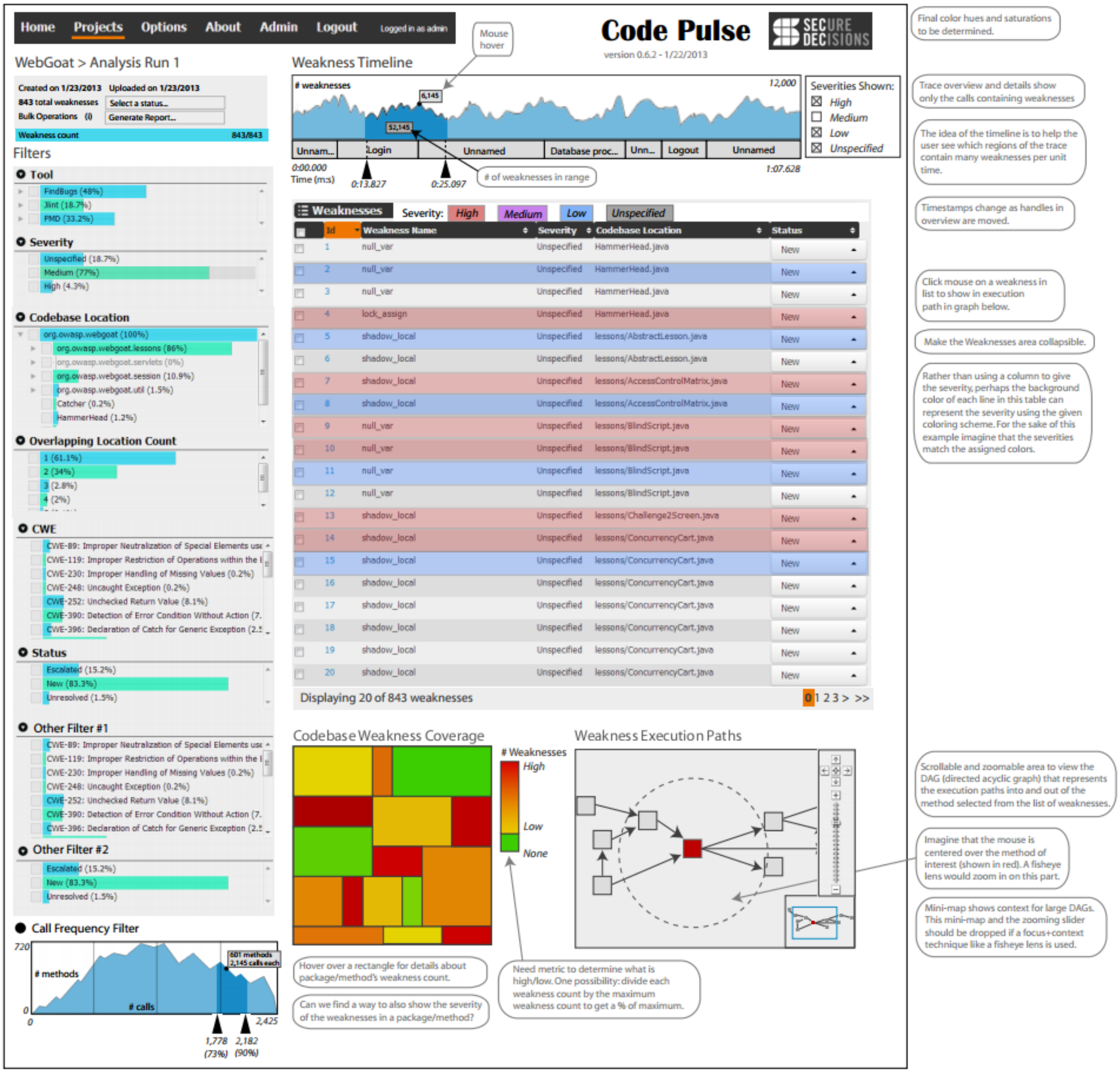

Figure 5. Layout and visualization exploration mockup

4.3 Development

This section details the development activity for Code Pulse starting with the system architecture and ending with an overview of the final user interface.

4.3.1 Architecture

There are two top-level components to the Code Pulse system architecture: the dynamic tracing component responsible for tracing code coverage of target applications at runtime; and the front-end user interface responsible for presenting the trace information in an easily digestible manner.

A key constraint of the dynamic tracing subsystem is the requirement to have minimal impact on the resources of the target application. To satisfy that constraint, the system had to be designed to perform minimal work in the same execution context as the target application and instead rely on another context to process the trace data. Therefore the dynamic tracing component was set up into two distinct pieces using a client / server model. The agent (client) runs in the same Java Virtual Machine (JVM) as the target application that will be traced. As the target application runs, the agent listens in on the execution and sends the traced information to the server for processing and storage. This high-level separation is shown in Figure 6.The separation in responsibility between the observer and data is key to limiting the impact on the traced application and reduce the footprint of the agent to the lowest possible condition. Note that nothing prevents the agent and server from running on the same machine, and in fact is anticipated to be a frequent use case.

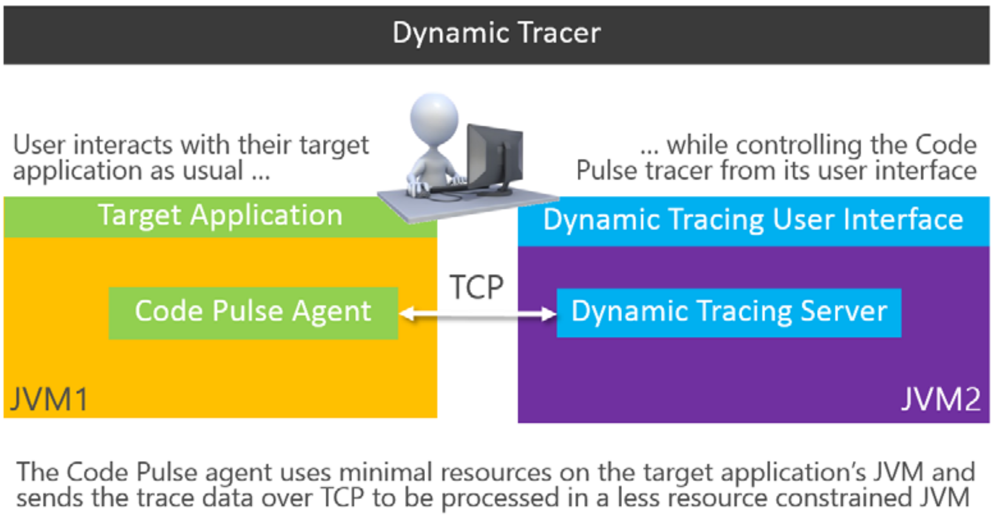

Figure 6. Dynamic tracing high-level component architecture

The Code Pulse tracing agent consists of only the components required to inject instrumentation into the application being traced and shuttle the data to the back-end for processing. The tracing agent communicates with the back-end over Transmission Control Protocol (TCP), sending the live trace coverage events upstream for processing and heartbeats containing current trace status, as well as receiving tracing commands resulting from front-end User Interface (UI) interactions.

Once the data is received on the back-end, it is consolidated and sent to the front-end for display purposes. This all happens on-the-fly, allowing the visualizations to be instantly updated with live data.

4.3.2 Technology Stack

Code Pulse makes use of a wide array of technologies. Leveraging existing solutions allowed us to focus our efforts on the areas directly impacting the Code Pulse capability.

4.3.2.1 Tracing Agent

Since the Code Pulse tracing agent is loaded into every application while it is being traced, a conscious effort was made to keep the volume of code and the number of third party dependencies as small as possible. Due to this concern, the tracing agent is written in pure Java with minimal third party dependencies.

We wrote a small common library to provide implementations for shared concerns between the back-end and the tracing agent. These concerns include configuration, data queuing, and the protocol used for communication.

The only third party dependency used by the tracing agent is the ASM bytecode manipulation framework[2]. This dependency provides the basis for the bytecode instrumentation required for tracing.

4.3.2.2 Back-End

The back-end is not loaded into other applications during tracing, allowing us to relax the code volume and dependency concerns. The back-end is written in Scala, and provides the functionality required for receiving and organizing trace data from, and control of, the tracing agent.

The back-end makes use of Slick[3] and H2[4] for data storage.

4.3.2.3 Front-End

The front-end user interface is implemented as a web app with the ability to run in a variety of Java servlet containers. There are two primary sub-components to the front-end – the server-side component running on the servlet container and the client-side component running in the web browser.

4.3.2.3.1 Server Component

The server component provides the heavy lifting for the front-end. It is primarily written in Scala, utilizing several helpful libraries, including:

- Lift[5] – provides the base web framework from which most of the Code Pulse front end is built.
- Akka[6] – provides an actor system framework, utilized by many of the data processing and updating tasks.
- ASM[2] – used for analyzing uploaded Java applications to build the code tree.

4.3.2.3.2 Client Component

The client component consists of the HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript rendered and run by the web browser. Several third-party components are leveraged for their rich functionality, including:

- bacon.js[7] – provides functional reactive programming, allowing for clean implementation of all the display elements that update in real-time.
- D3.js[8] – provides a framework for data-driven DOM manipulation, allowing rich visualizations that perform well.
- jQuery[9] – primarily utilized for event handling and DOM manipulation.

4.3.2.4 Distributable Application

Despite Code Pulse being developed using web application technologies, it is packaged as a desktop application. To enable that, we leverage node-webkit[10], a wrapper around Google's Chromium Embedded[11] project providing a customizable browser application. The end result is a stand-alone downloadable application that behaves as a desktop application despite being written as a web application.

4.3.3 Dynamic Tracer

At the heart of Code Pulse is the dynamic tracer. Dubbed *bytefrog*, this is the piece of the system responsible for the live application trace data that is used for providing coverage information.

The tracing agent is a standard Java pre-main agent. It is inserted into the application being run via the `javaagent` command line argument, at which point, it is able to hook into the JVM instrumentation API.[12]

The dynamic tracer records events as they occur during program execution. The primary events of interest include when methods are entered and exited and when exceptions are thrown. These events can then be processed to gain insight into which pieces of code were executed.

4.3.3.1 Instrumentation

To collect these events, callbacks must be inserted into the traced application at all points of interest. The process of inserting these callbacks is known as *instrumentation*. A class transformer is registered with the JVM, allowing classes to be transformed as they are loaded for execution.

During this transformation process, the required callbacks are inserted. As mentioned, the primary points of interest are points where methods are entered, points where methods are exited, and anywhere an exception is thrown. Effort was made to keep these injected callbacks as minimal and quickly executing as possible, with the end goal of slowing down the traced application as little as possible.

4.3.3.2 Data Collection

The callbacks inserted during instrumentation insert data messages into a queuing system. This queue is processed using several threads, sending chunks of data upstream to be processed.

An enormous amount of trace data is generated, at a rate faster than it can be processed. The queuing system has a finite capacity, serving as a buffer. This means that applications exhibiting short bursts of activity will see a minimal slow down (e.g., web applications). When larger bursts of execution occur, the buffers fill up, and then execution will slow down to the rate at which the resulting data is processed.

4.3.3.3 Data Protocol

Data is encoded before it is sent for processing. A special protocol has been designed to allow for all necessary communication with the tracing agent, while at the same time keeping all communication compact. The protocol consists of a series of control and data messages, a sample of which can be seen in Figure 7.

Method Entry Message

Description

The `method entry` data message tells HQ a method entry has occurred.

Format

```
[1 byte: type ID (20)][4 bytes: relative timestamp][2 bytes: current sequence][4 by
tes: method signature ID][2 bytes: line number][2 bytes: thread ID]
```

The message type ID for this message is `20`. Message content will be the timestamp offset (32-bit unsigned, milliseconds since start of tracing), followed by the current sequence ID (16-bit unsigned), followed by the method signature ID, followed by the source line number (16-bit unsigned), followed by the thread ID. Message size is fixed at 15 bytes.

Figure 7. Snippet from the Code Pulse message protocol specification

4.3.3.4 Data Processing

The actual processing of the data happens within Code Pulse. After data is sent over from the tracing agent, it gets queued up for data processing. The data is sorted into the proper order and analyzed sequentially. This, essentially, creates a replay of the events in an environment where they can be processed to fully glean code coverage data.

The constructed code coverage data then makes its way into a database and to the front end UI. This entire process happens on-the-fly, giving near real-time display of code coverage to the user.

4.3.3.5 Future Work

One major area of potential future work lies in the improvement of the dynamic tracing pipeline. The current implementation is extremely powerful, and with this robustness comes a large amount of data that is unnecessary for code coverage purposes. A live play-by-play stream of data, as is provided now, is unnecessary for the task of determining code coverage. Adding the data required to provide line-level coverage details would greatly increase the data flow, leading to definite performance issues.

Fortunately, it is viable to overhaul the tracing pipeline to focus on the specific needs of tracking code coverage. If the tracing agent itself knows how to track code coverage, this task can be immediately performed, eliminating the need for recording every method entry and exit. The result would be *much* less data gathered and less work done by instrumentation callbacks – which means reduced overhead. With less data to transfer and process, the entire process becomes faster and less impactful on the application being traced.

4.3.4 Alpha Prototype

The alpha prototype was completed on 12/9/2013 with features geared around the DESA use cases.

4.3.4.1 Dynamic Tracer with Graphical User Interface

To control the instrumentation capability, a user-facing application was created following the wireframes created in Task 2. Figure 8 shows the configuration screen of the Dynamic Tracer developed during this reporting period. It was implemented with JavaFX and served as our primary mechanism to collect dynamic trace data for the alpha prototype. Files generated by the tracer may be uploaded as part of a Code Dx analysis (or added to an existing Code Dx analysis), providing Code Dx with the information it needs to make correlations between dynamic and static analyses.

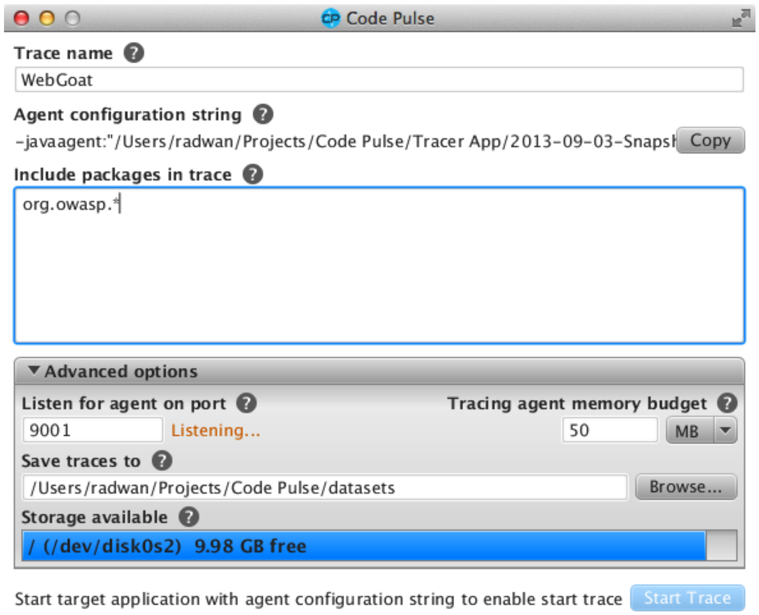

Figure 8. Dynamic Tracer trace configuration screen

4.3.4.2 Dynamic Trace Correlations

After using the dynamic tracer to create traces, the next step in using the traces to improve the weakness triage process is to correlate them to the weakness information collected from the static analysis tools. In doing so we faced some challenges.

The first challenge was the magnitude of the dynamic tracing data. While we'd managed to compress the information significantly on disk to reduce disk storage, the data volumes were still significant for correlation processing. To overcome this, we decided to take the approach of generating a summary form of the trace data to satisfy the immediate requirements of the Code Dx trace data integration. The tradeoff is that the data set stored does not include the full rich trace information, but on the other hand, the data storage and correlation speeds were improved. As we decide to expose more of the trace data, we'll change the stored data to satisfy the new requirements, provided it does not prohibitively hinder the storage and correlation speeds.

The second challenge was in the different viewpoints between time-based and weakness-based data. The obvious pivot point between the two types of data is the source code location, which is exactly what we opted to use as our primary correlation mechanism. However, the method calls observed in the traces were time-based and the weakness information is typically statically associated with a particular method. Correlating purely based on the source code location would lose the valuable time component of when these calls were made. To overcome this, our approach was to compromise on the fine-grained accuracy of the method calls and instead aggregate the time-based data into correlated time buckets. For instance, if a trace lasted for 100 seconds, we'd create 100 time buckets, one per second, and associate the method calls per time bucket. Therefore we'd be able to identify weakness-affected methods based on their general timeline as opposed to specific timestamp. The time length of each aggregate bucket is a variable that impacts both accuracy and the amount of data stored. The larger the time bucket interval, the less data is stored with the side-effect of having less accurate timing information for weakness correlation.

The Code Pulse dynamic tracer was tweaked to store only the weakness-correlation necessary data, and the Code Dx data processing pipeline was upgraded to accept dynamic traces and correlate them with static analysis tool results. With the correlated data, Code Dx was able to show how often and when in the trace execution timeline weakness-affected source code is called.

4.3.4.3 Filtering Weaknesses Based on Trace Data

Since filtering plays a pivotal role in the triage workflow, the alpha prototype added two dynamic trace-related filters to Code Dx.

The first of these filters was for the traces and their user-created segments. A new "Trace Segments" filter was added to the analysis run page. In the filter, a list of traces and segments within each trace is displayed to the user. The segments are displayed in a tree hierarchy reflecting the nesting order created by the user during the dynamic trace collection. Users can select one or more traces, or expand the traces and select one or more segments. Whatever selection the user makes, the whole page filters on just the subset of weaknesses that match. In this particular case, the matching weaknesses are the ones in weakness-affected source that was executed in that timeframe. If an entire trace is selected, then the weaknesses observed during that trace are matched, or alternatively just the weaknesses observed during a specific time-range described by a trace segment. Figure 9 shows the new filter populated with the information from a single trace.

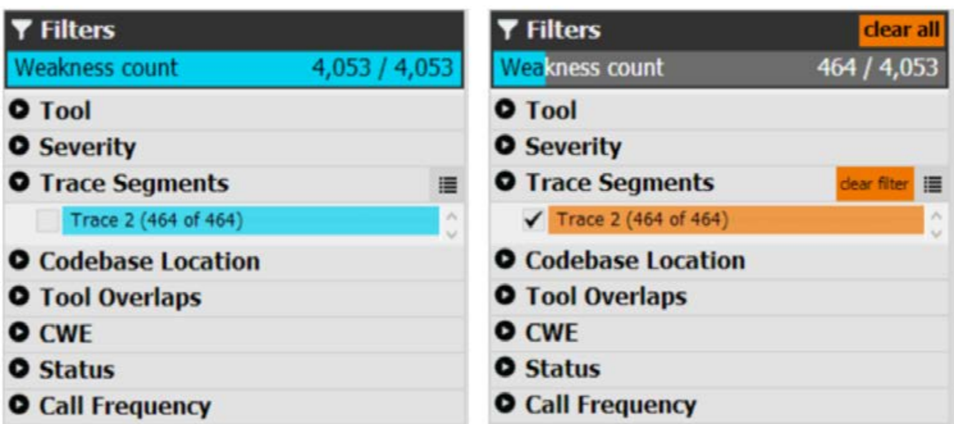

Figure 9. Trace segments filter showing before-and-after states of filter selection

The second new filter addition was for the call frequencies as observed from the dynamic traces. Call frequencies were broken down into percentiles and split up between 10 groups ranging from the least to the most called methods. These groups are displayed to the user in the filter and can be selected in any combination. Figure 10 shows the breakdown and the weakness matches for the 90th percentile and greater called methods. This leads to interesting analysis scenarios where users can chose to prioritize the least or most called methods, including anything else in between.

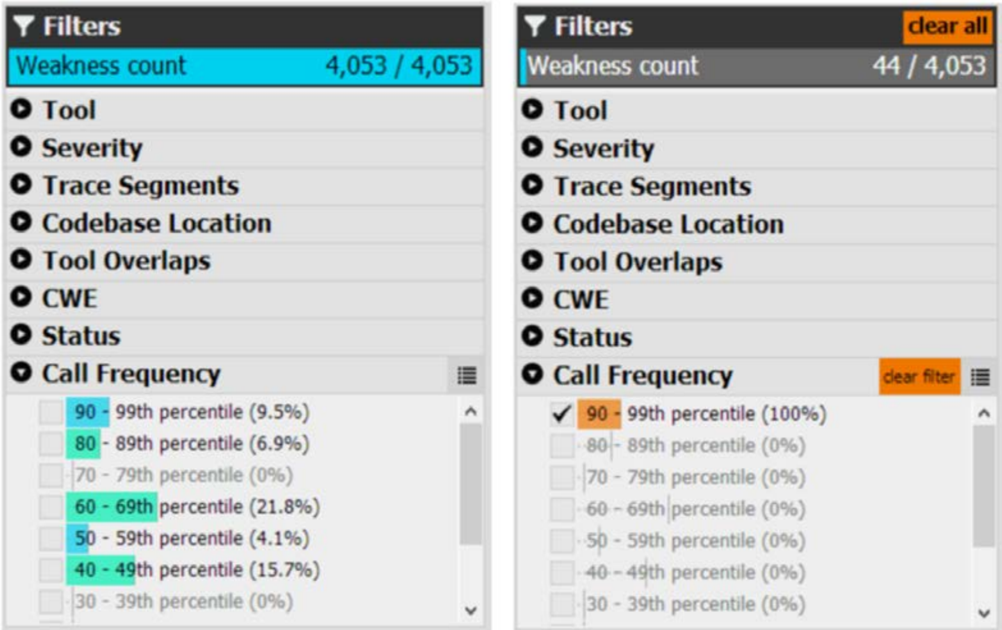

Figure 10. Call Frequency filter showing before-and-after states of filter selection

4.3.4.4 Trace Details Page

We added the Trace Details page to Code Dx, for each analysis that included trace data. The new page prominently features a treemap visualization of the analysis's codebase. The treemap is used to easily identify where correlations were made between static and dynamic analysis. The treemap shown in Figure 11 uses red to indicate methods that only had static weaknesses, blue to indicate methods that were traced but had no static weaknesses, and purple to indicate methods that were both traced and had static weaknesses (indicating a correlation between static and dynamic analysis).

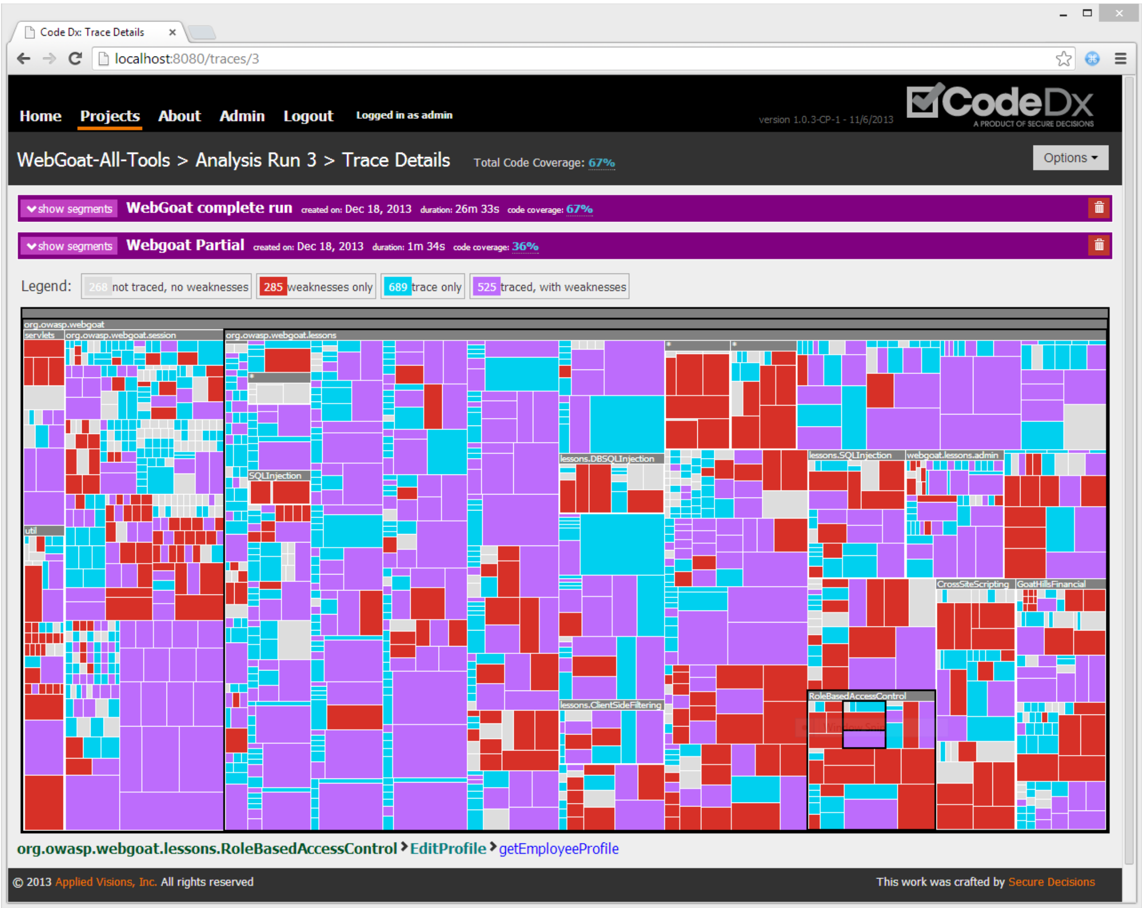

Figure 11. Trace details page from the alpha prototype

4.3.5 Final User Interface

Once the idea of Code Pulse as a live code coverage tool crystalized, we set out to create a new user interface geared towards these new use cases. The new user interface was developed separately from Code Dx, becoming a standalone application. This section describes the features that we developed, and some of the challenges that we faced while creating the new interface.

4.3.5.1 Code Coverage Tool Development

A lot of effort went into the backend to integrate the tracing capability into a real time visualization centric user interface. This differed from the original asynchronous nature of the collection and analysis. As a result a number of significant extensions were needed to our tracing infrastructure to support streaming data coming in at high volume to a distilled set that would be more manageable by a responsive user interface.

Although we didn't start from scratch with the user interface since we had a lot of key elements already done from the Code Pulse alpha, putting it all together in a cohesive, efficient, and maintainable whole was the primary challenge while creating the final user interface. The result can be seen in Figure 12.

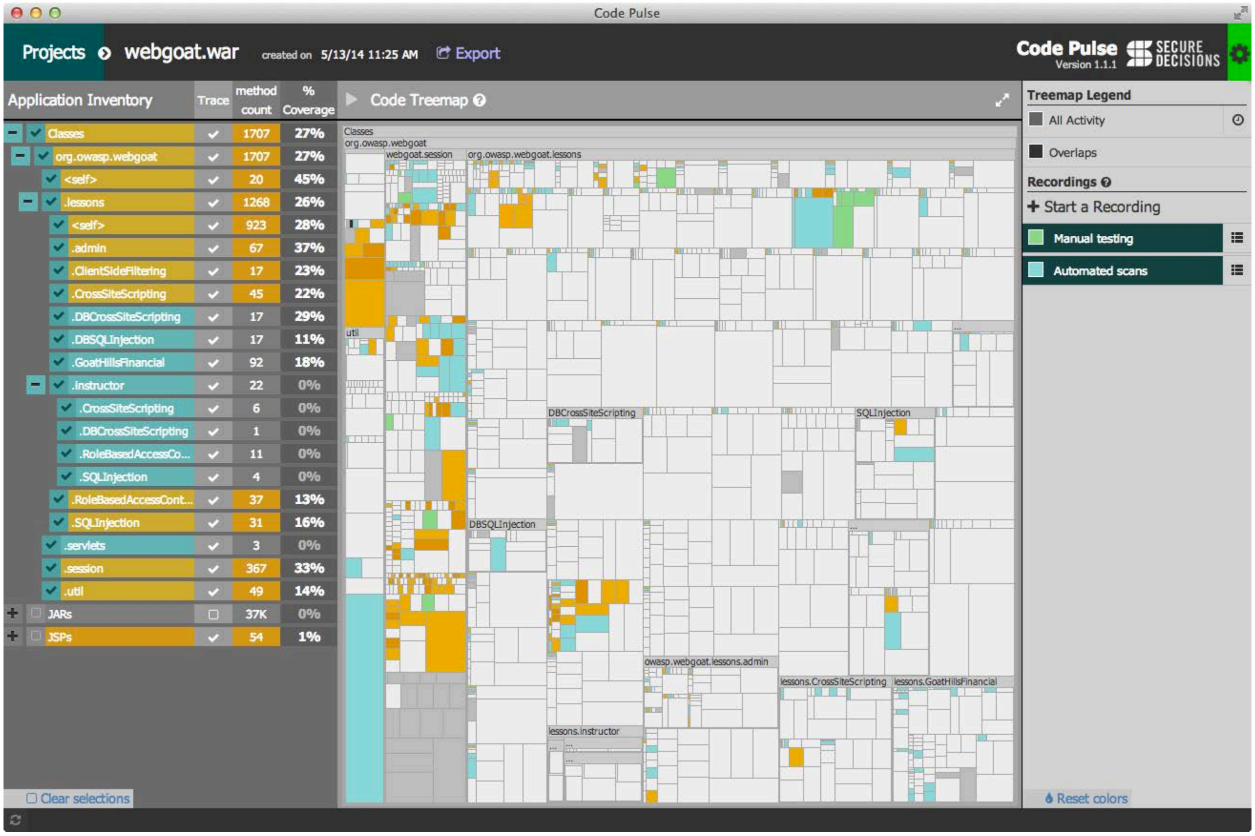

Figure 12. Final Code Pulse code coverage user interface

4.3.5.2 Treemap Scalability

We've used treemaps in the past to represent codebase hierarchies and knew that while they are incredibly effective tools to communicating hierarchy, codebases tend to result in tens of thousands or even hundreds of thousands of nodes when explored at the method level of detail. With the exception of 3D based graphics libraries, rendering that many nodes graphically is incredibly resource intensive and ultimately impractical. Switching to a 3D based graphical library was too risky a proposition for us at the current stage of the project so we explored alternative approaches to limiting the number of objects rendered simultaneously.

We explored reducing the level of detail to focus on the class level instead of methods. However, that significantly reduced the utility and fidelity of the coverage information. Looking at coverage information at a class-level aggregate significantly hampered the understanding of how much of the code was actually covered. Having determined that codebase methods was the requisite level of detail, we started exploring other techniques to limiting the number of methods represented in the treemap at a given point in time. We ultimately settled on showing a tree view of the codebase's packages and limiting treemap viewing to package selections. There were tradeoffs to this approach. Seeing the full treemap is a much better visual at-a-glance indication of the coverage than a tree view based approach. However, this was ultimately a solution that could scale with the application and was the adopted approach.

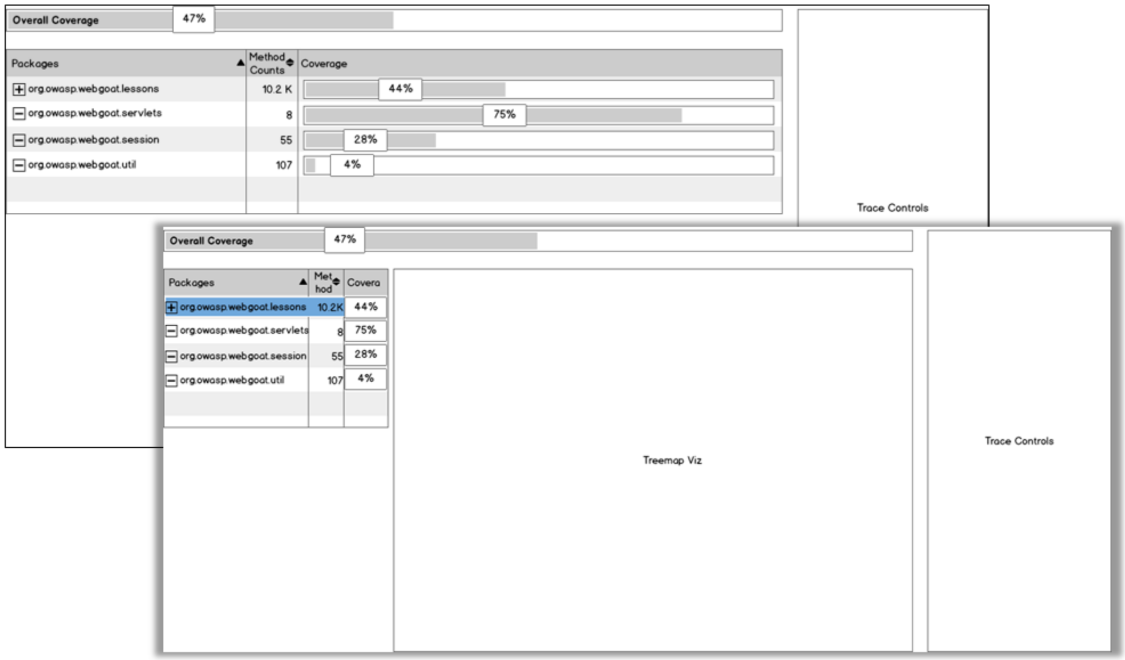

Figure 13. Mockups of package tree view to control what is represented in the treemap

Figure 14 shows the final working version of the tree view of the codebase's packages alongside the corresponding treemap visualization. We divided the codebase into three abstract groupings; one for application classes, a second for Java ARchive (JAR) files embedded the application (such as third-party libraries), and a third for Java Server Pages (JSP) files.

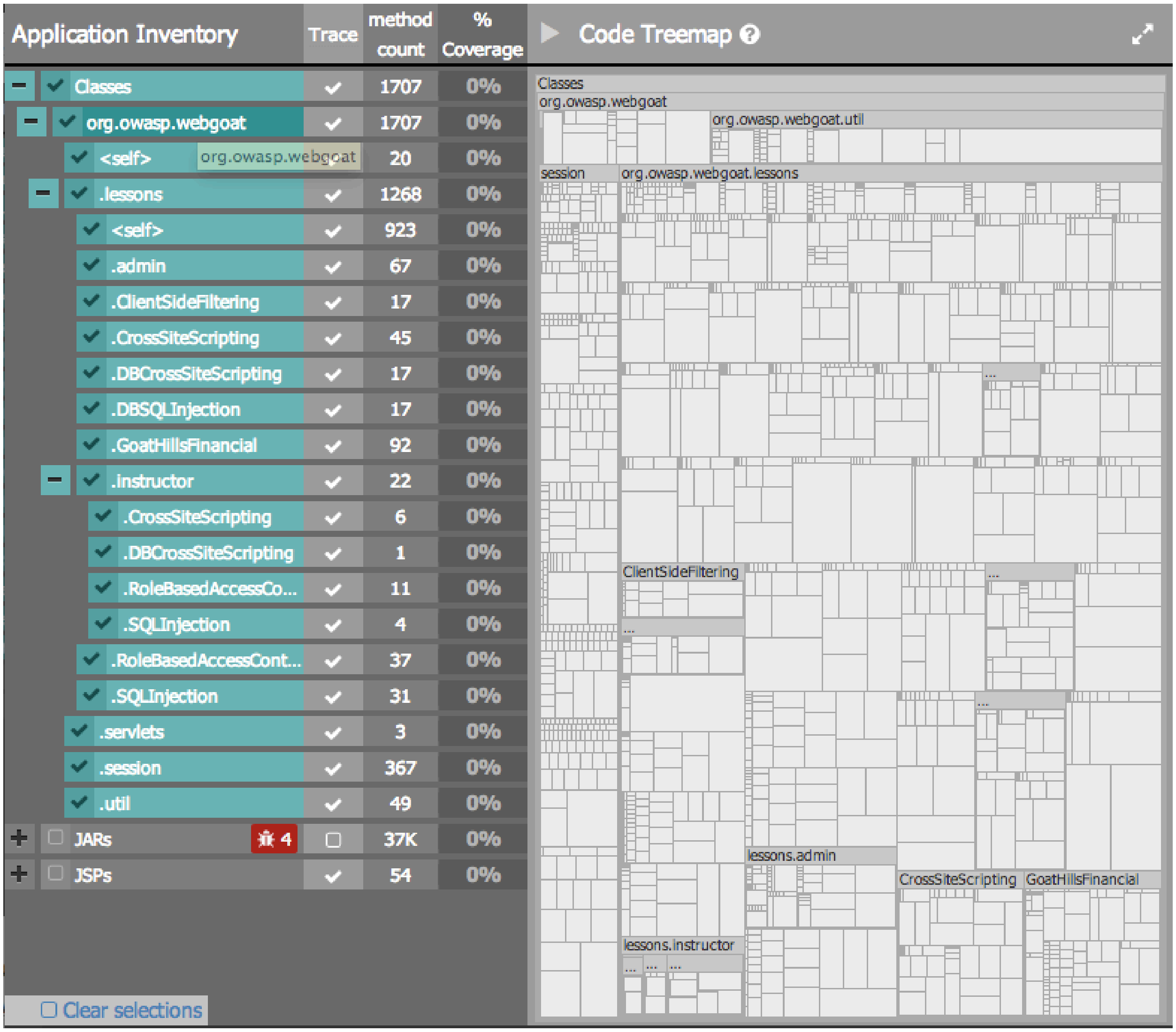

Figure 14. The Application Inventory tree view next to the codebase treemap

In Figure 14, the "Classes" subtree is selected, causing the codebase treemap to display only the hierarchy for that subtree, excluding any objects from the "JARs" and "JSPs" trees. Selection is indicated by a teal highlight and a checkmark to the left of the objects' names.

4.3.5.3 Instrumentation Filter

We found that the "JARs" grouping mentioned in the previous section tended to contain a large majority of classes, while application classes were generally the primary focus of tracing for code coverage. Our initial approach of tracing all detected classes suffered due to the performance implications of adding instrumentation to a class. To remedy this, we added the instrumentation filter to the user interface which allows users to pick which classes should generate trace data at runtime. The "Trace" column in Figure 14 displays checkboxes; by clicking any of these checkboxes, users can toggle the instrumentation filter on or off for the corresponding groupings.

4.3.5.4 Recordings

For the final user interface, we created the concept of "Recordings," which marks a segment of time started and ended by the user during which trace activity will be monitored. Code coverage information is collected for each recording, which lets users mark different execution sessions with recordings. For example, a "startup" recording could be created to mark the startup sequence of the traced application, then ended when that sequence finished. Recordings are independent of one another, meaning that stopping and starting one recording does not have any bearing on the state of other recordings.

When a recording is selected, any methods traced during that recording will be highlighted in the treemap with a corresponding color. This makes it easy to see at a glance which methods were covered, during which recordings. The user can change the color of any recording at any time by clicking the color swatch next to the recording's name.

We created two special pseudo-recordings to satisfy special cases; the *Activity Ticker* and the *Overlaps* recording. Their user interface is similar in appearance to recordings, but their interaction behaves differently; they cannot be stopped, and they cannot be selected in the same manner as recordings. Both pseudo-recordings are always implicitly selected; they always affect the coloring in the treemap.

The *Activity Ticker* represents trace coverage data between some variable starting point and the present. By default, the *Activity Ticker* represents all trace data, as if its starting point was the beginning of the trace. The starting point may be modified to show the latest 10 seconds or latest five minutes, for example. This is accomplished by selecting the appropriate time window from the drop down menu shown in Figure 15. Methods that were traced during the *Activity Ticker*'s time window will automatically be colored in the treemap, using its current color. When other recordings are selected, this color will fade slightly to avoid drawing focus away from the selected recording's colors.

The *Overlaps* recording dictates the color displayed when multiple recordings are selected and a method was encountered during more than one of them.

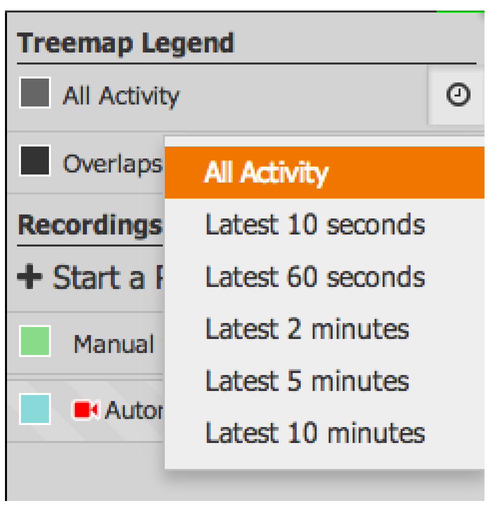

Figure 15. Changing the "All Activity" recording to a time segment

4.3.5.5 Trace Connection

We created a simple workflow for connecting traces to Code Pulse. When a user runs their application with the Tracing Agent attached, the agent will attempt to connect to Code Pulse. If successful, the message shown in Figure 16 will appear in the upper-right corner of the UI, notifying the user of the new connection.

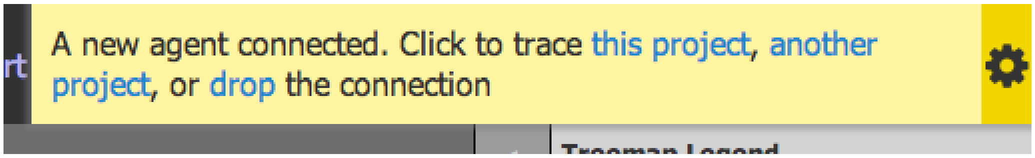

Figure 16. Agent Connection Message

The user can then click any of the links in the connection message to complete or drop the trace connection. Once completed, the trace will immediately start generating data, and the user interface will immediately start displaying that data.

While the trace runs, the Tracing Agent streams the traced application's activity to Code Pulse's back end, which processes the event data into coverage data. Coverage data is regularly sent to the front end.

The front end applies an orange flashing effect to draw attention to methods as they are called. When a method is called, the corresponding node in the treemap and its corresponding parents in the Application Inventory will turn orange, then gradually fade back to their original colors. Figure 12 shows this effect in action.

As the total trace coverage set updates, so do the colors of nodes in the treemap. For each selected recording, if a method was called during that recording, its corresponding node in the treemap will display the color of that recording.

4.3.5.6 Dependency Check Integration

We integrated with OWASP Dependency Check[13], displaying which JAR files in the application contained known Common Vulnerabilities and Exposures (CVEs). If a JAR contains one or more CVEs, the corresponding entry in the Application Inventory will display a red "bug" icon; clicking this icon brings up the Dependency Check result for that JAR file. Clicking the icon on the top-level "JARs" group brings up the full report which covers all JARs in the application, shown in Figure 17.

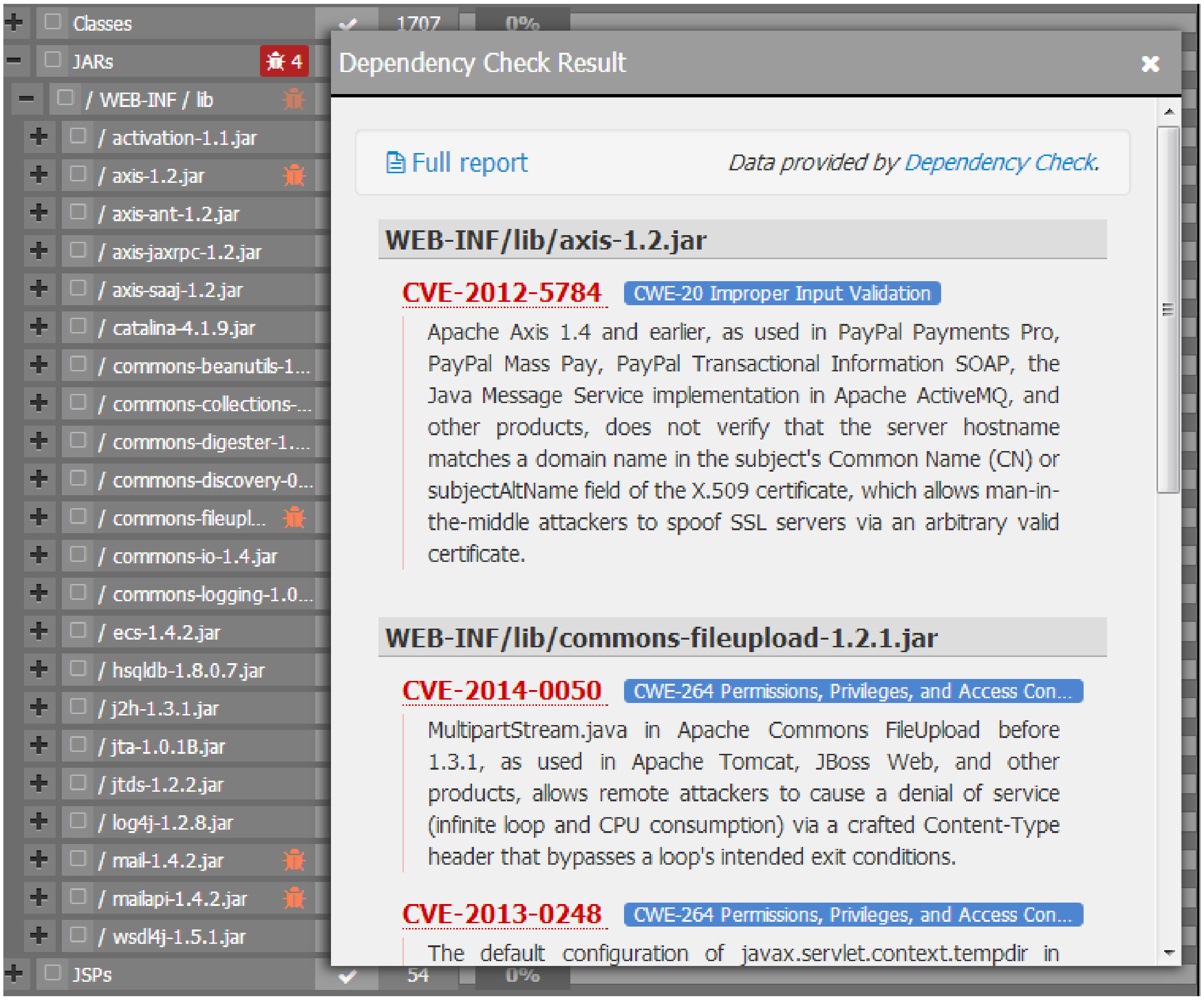

Figure 17. Dependency Check report for third-party JAR files

4.4 Testing and Evaluations

This section describes our testing approach and demonstrations to potential users and interested parties. This work was done to ensure what we developed was functioning as designed from a quality perspective, demonstrate it to potential users, and evaluate its operation to determine if it's meeting user's needs effectively.

4.4.1 Testing

Testing of Code Pulse was conducted from the start of development through alpha, beta, and public release milestones. Details of our testing methodology is in our Software Test Plan deliverable, CP-STP-0001. A summary of this approach is outlined in this section.

Code Pulse testing included unit, integration, system, user acceptance, functional, security, reliability, performance, and portability testing. Any testing defects found were cataloged in our defect tracking system for remediation. All critical known defects have been resolved.

4.4.1.1 Test Environment

Code Pulse testing was conducted on existing Secure Decisions hardware at its Northport and Clifton Park, NY facilities in addition to evaluator sites. Testing was conducted on the three key components of the Code Pulse software: the dynamic tracing agent monitoring target applications; the server component collecting the events observed during dynamic tracing; and the user interface elements to analyze the results.

A diverse set of application was used for testing. For our alpha version (before we decided to focus on web applications), we tested a mix of web and desktop application, ranging from a few thousand line of code to several hundred thousand.

For our beta testing we added the OWASP Zed Attack Proxy (ZAP) DAST tool to measure its testing coverage, and several test application including BodgeIt, WAVESEP, WebGoat, and Jenkins.

4.4.2 Demonstrations and Evaluations

Our agile development approach and close interaction with end users caused us to go through several iterations and changes in direction, but the end result we feel is something that provides unique value and addresses a major need in application security testing today.

Demonstrations and evaluations went through two phases. The first was for our alpha version, and the second was with our beta version that focused on code coverage.

When our alpha release was complete, we performed an extensive round of demonstrations and evaluation sessions with a variety of potential users. The individuals or groups we met with were both internal and external to the company and selected for their subject matter expertise in software assurance or for their likelihood to be potential users of the tool, see Table 6.

Table 6. Code Pulse alpha version evaluations

Evaluator(s)	Company/Organization	Date
Ken Prole, Anthony DeMartini	Secure Decisions	12/6-13/2013
Drew Buttner	MITRE	12/18/2013
Steve Noel, Ganu Kini, Aaron Temin, et. al.	MITRE	12/19/2013
Lenny Halseth	Secure Decisions	12/20/2013
Michael Rosenstein	Secure Decisions	12/27/2013
Jim Manico	Independent	1/16/2014
Jerry Hoff	WhiteHat	1/17/2014

The evaluations conducted in December were focused on the Code Pulse alpha and were structured to start off with a quick briefing on the project concept and the use cases. This was followed by a demonstration of the alpha version with targeted discussions on all three primary use cases: tracing; prioritization; and remediation. The following is a summary of the feedback we got from the evaluations grouped per primary use case:

- Tracing use case:
    - Evaluators thought that the tracing user interface was straightforward and did not perceive the agent hook requirements to be a hurdle to adoption or usage.
- Prioritization use case:

    - A primary concern for the evaluators was that they did not think it was realistic to be able to generate a trace that captures the complete execution profile of an application. Thus, they did not think that basing the prioritization on the trace data was a good recourse.
    - Due to how traces were likely to be captured, they were likely to be reflective of typical execution patterns. However, attackers are likely to be prompting unusual execution patterns that may not be observed in typical traces.
    - Evaluators did not feel comfortable (temporarily) ignoring statically observed weaknesses even if the weaknesses were not observed in code that's part of the attack surface.
    - There was agreement that for performance impacting weaknesses there is value in using call frequency to prioritize weakness remediation.
    - There was interest in understanding which code was less frequently traversed to prompt closer review and assessment.
    - The treemap visualization was appealing to the evaluators and resonated with them.
- Remediation use case:
    - There was unanimous agreement amongst the evaluators that the best place to view information pertaining to remediation should be in the IDE.
    - There was interest expressed in being able to automatically determine if a vulnerable code path as detected by the SAST tool(s) is feasible as observed from the traces.
    - Evaluators generally connected with the described remediation use case and expressed interest seeing the call graph generated from the traces but when pressed to describe the value they would get from the call graphs, there were no clear answers. It was difficult to assess the real versus perceived value from this round of evaluations.

Generally speaking the reception of the Code Pulse alpha was lukewarm. The evaluators were intrigued by certain concepts and features, however, overall they struggled to identify with the key premise that a trace profile has utility in the prioritization process. As we discussed these results, we realized that there was overall interest and enthusiasm for the parts of the system dealing with code coverage. This prompted us to try an experiment and throw together a quick proof of concept to further explore the idea.

The two evaluations we conducted in January 2014 were specifically requested to get a penetration testing perspective on our prototypes, particularly of the proof of concept code coverage prototype that we'd developed. Their feedback on the Code Pulse alpha echoed what we'd heard previously. However, when we showed them the code coverage proof of concept prototype they were incredibly excited and enthusiastic at the concept. That's when we realized that we were on to something and focused on future work on the code coverage use-case.

For our beta code coverage focused release in April 2014, we again sought out subject matter experts to get their feedback, particularly on the usability and utility of what we developed. Table 7 lists these in-person evaluation session that we conducted.

Table 7. Code Pulse beta version evaluators

Evaluator(s)	Company/Organization	Date
Ken Prole	Secure Decisions	3/21-4/8/2013
Jim Manico	Independent	4/10/2014
Dave Wichers	Aspect Security	4/15/2014

The feedback we received from these evaluators was very positive and all felt the user interface was intuitive and the use-cases and value of Code Pulse's visual representation was really important for penetration testers who are currently unaware of the testing coverage they are achieving.

Much of the feedback we received during these evaluations was added to our 1.0 or 1.1 release, including making it easier to start tracing, usability tweaks, performance optimizations, and visibility into library dependencies and their security state. Other feedback we received but did not implement was added to our product roadmap.

4.5 Transition

The following is the results of our market and competitive analyses along with an outline of our transition strategy for Code Pulse. A detailed description of our transition research and strategy was submitted on 5/30/2014 in the Technology Transition Plan report.

4.5.1 Market Analysis

Trends in the cyber security market consistently indicate increases in the frequency and costs of cyber attacks. In October 2013, the Ponemon Institute study of cyber attacks and their financial impacts indicated that the average organizational cost of cyber crime is $11.6 million per year, representing a 26% increase, or $2.6 million over the previous year.[14] When seeking to determine the fundamental cause of such attacks, it is estimated that 90 percent of reported security incidents result from exploits of defects in the design or code of software.[15] Organizations are faced with the added pressure to comply with regulations such as Health Insurance Portability and Accountability Act (HIPAA), Payment Card Industry Data Security Standard (PCI-DSS), and Federal Information Security Management Act (FISMA),[16] and to control Information Technology (IT) budgets despite the threat of data and financial losses from cyber incidents. Those organizations that minimize exploitable software flaws in their enterprise software, particularly those vulnerabilities related to their industry's security standards, reduce their exposure to risks from cyber security attacks and risk of sanctions for non-compliance with industry standards.

Our analyses show that the software security market continues to grow at a healthy pace. According to a Gartner Group report, worldwide security software revenue totaled $17.7 billion in 2011, a 7.5% increase from 2010 revenue of $16.4 billion. Revenues continue to increase steadily within companies providing software assurance solutions such as IBM/Rational, experiencing increases in revenue between 1.8% and 5.0% from 2008 to 2011.[17] Klocwork announced the largest quarter of bookings in company history for Q3 2012, with annual growth of 18%, driven by demand in the US aerospace and defense sector for software security.[18]

4.5.2 Competitive Analysis

The competitive field of software quality assurance testing tools is quite extensive and mature, and in some cases saturated. Given the discriminating features of Code Pulse, there exist opportunities for Code Pulse to establish a presence in selected testing domains such as:

- Code coverage analysis testing
- Penetration testing,
- Profiling, and
- Integrated Applications Security Testing (IAST)

A review of the code coverage analysis tools market reveals an extensive and mature field of commercial, free, and open source tools ranging from stand-alone coverage analysis utilities to extensive quality assurance testing tool suites. Several of these have reached a significant maturity level, and have been actively developed and maintained for more than a decade. Consequently, test and quality assurance engineers seeking code coverage analysis tools have a reasonably robust choice of tools to select from. What is interesting to note amongst the set of tools in the field is that the ones with staying power are those that have developed plugins for the major software development frameworks, such as Eclipse and Visual Studio. It is a market filled with abundant plugins, and it appears that for a product to remain a contender in this domain, it must offer plugins to several of the available development and testing frameworks.

An analysis of the penetration testing tools market reveals a fairly well populated and mature field of commercial, free, and open source tools. The tools within this domain consist of various exploitation tools, password crackers, website scanners, and vulnerability scanners. Still other tools provide more comprehensive and in-depth frameworks for testing and vulnerability remediation and management. Several tools have met with success in the marketplace and have had significant staying power, such as Metasploit, QualysGuard, Retina, and in the open source world, OWASP ZAP. Distinctive within the penetration testing tool space however is the scarcity of visualization tools, particularly as they relate to vulnerability scan analysis. Many applications support a command line interface, especially open source tools. A small selection of tools provide Windows or web user interfaces including dashboards. However, the few vendors that do provide dashboards (Acunetix, Nexpose, and LanGuard, for example) use them to manage patching or organize remediation initiatives. None appear to use a dashboard to assess and display the coverage of their vulnerability scans, especially in real time, nor visually identify problem areas in code modules. This particular niche within the pen-testing tool environment can present an opportunity for Code Pulse to enter this market. The Code Pulse combined dashboard and visualization for presentation of real time assessment of vulnerability scan coverage appears to complement existing tools, and can be a unique advantage in the penetration-testing tools market.

An analysis of the profiler tools market reveals a well- populated and mature field of commercial, free, and open source tools. The applications within this domain typically consist of a collection of tools to tune computer Central Processing Unit (CPU) and Graphics Processing Unit (GPU) performance, memory error detectors, thread error detectors, cache and branch prediction profilers, heap profilers, stack overrun detectors, branch prediction profilers and call-graph tools. The more advanced tools provide sorters, filters and visualizers to present results and provide insight into performance bottlenecks. Commercial applications such as the ANTS Performance Profiler and open source applications such as Valgrind provide comprehensive and mature profiling solutions, complete with dashboard and visualizations for presenting performance metrics. The area where Code Pulse could possibly enter this market would be by providing visualization support of application execution results, visually identifying performance bottlenecks for the lower end commercial tools that lack such a capability, such as ACTime Pro. There are however few profiling tools that lack some sort of visualization capability. The market is replete with robust profiling tools, making the transition for Code Pulse with it current capabilities a challenging one.

An analysis of the Interactive Application Security Testing (IAST) tools market presents a relatively new and immature market. There are few players actively involved in the development of IAST tool solutions that combine dynamic and static techniques to improve the overall quality of software application testing results. The goal of these tools is to provide an inside-out view that complements the outside-in view of a purely DAST solution – for example, identifying the specific line of code where a security vulnerability occurred, or providing detailed visibility into code coverage.[19] No vendor appeared upon the IAST scene prior to 2012, and consequently no vendor has necessarily established itself as a leader in the IAST space, presenting a potential opportunity for Code Pulse.

4.5.3 Transition Strategy

Just as the project direction shifted from targeting the DESA approach to settling on the penetration testing code coverage approach, the transition strategy evolved with the project. Although we considered a number of potential avenues for the Code Pulse transition, the current key elements of our transition plan are listed as follows:

- Code Pulse was released as an open source tool to both benefit the application security community as well as serve as a marketing vehicle for Code Pulse itself and our broader Software Assurance work at Secure Decisions. Code Pulse is available on github.[20]
- Code Pulse was made an OWASP[21] project in April of 2014. OWASP is large application security community with a membership roster of over 42 thousand members. This will provide Code Pulse with a large audience out of the gate to help encourage adoption of the tool. In addition, OWASP has a large vibrant project inventory of 160+ projects with the strong potential for synergistic activities. We've already started down that road with the integration work we did with OWASP Dependency Check outlined in section 4.3.5.6.
- We will continue to explore the potential to release a "Pro" version of Code Pulse targeting professional penetration testers that have already adopted the open source community edition of the tool and seek added capabilities. This version of the tool would be provided for a licensing fee and would provide a revenue stream to support the tool's maintenance and continued development.
- The Software Assurance Marketplace (SWAMP) remains a strong potential transition site for Code Pulse. At the time of this report's writing we engaged the SWAMP team and demonstrated the tool to them. Although they expressed interest, it is still early for them to start exploring DAST related use cases and as such we shall revisit this in the future.
- A strong web presence serves as an anchor to point Code Pulse newcomers at and serves as a marketing tool. Along-side the version 1.0 launch, we also launched the official Code Pulse website at http://code-pulse.com.

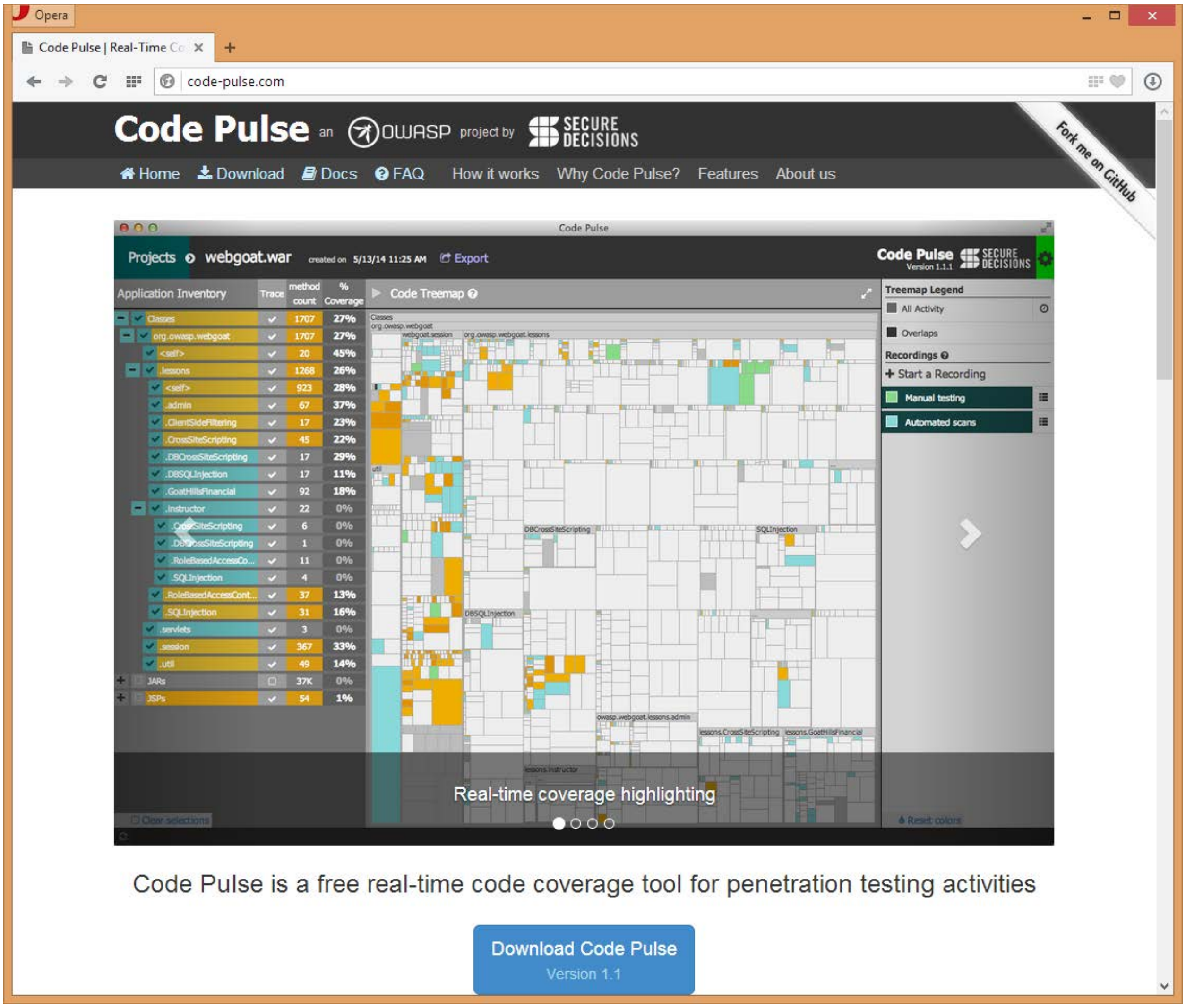

Figure 18. Code Pulse website

- We've increasingly adopted social media to help promote Code Pulse, primarily focusing on Twitter, with plans to expand the Code Pulse presence at Facebook and LinkedIn.
- We plan to seek speaking engagements to spread the word about Code Pulse and help increase the tools' adoption. Being part of the OWASP community has helped with that, although we do plan on casting a broader net beyond OWASP to help promote the tool.

5 CONCLUSIONS

Of the lessons the learned in the Code Pulse effort, perhaps the most important one is to involve evaluators as early as possible. The feedback we received from the subject matter experts played a defining role in the direction of the project, and helped shape its final form. Code Pulse has been released as an open source tool available to the application security community to help improve their penetration testing processes. It is an OWASP project and has an active network of security professionals to lean on as it slowly cracks out of its R&D cocoon. Code Pulse 1.0 was release on May 2nd, 2014 and has since been updated twice to add new capabilities, improve the tool's usability, and address uncovered bugs. Code Pulse has been downloaded a total of 200 times since version 1.0 and continues to be actively promoted.

While we write this at the completion of the period of performance there is a long list of transition and capability enhancement activity we aspire to achieve with Code Pulse. In its current form and as it continues to evolve Code Pulse adds an effective tool to the penetration tester toolbox.

6 REFERENCES

[1] http://www.sable.mcgill.ca/starj/

[2] http://asm.ow2.org/

[3] http://slick.typesafe.com/

[4] http://www.h2database.com/

[5] http://liftweb.net/

[6] https://typesafe.com/platform/runtime/akka

[7] https://github.com/baconjs/bacon.js/tree/master

[8] http://d3js.org/

[9] http://jquery.com/

[10] https://github.com/rogerwang/node-webkit

[11] https://code.google.com/p/chromiumembedded/

[12] http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

[13] https://www.owasp.org/index.php/OWASP_Dependency_Check

[14] "2013 Cost of Cyber Crime Study, United States", Ponemon Institute, October 2013, http://media.scmagazine.com/documents/54/2013_us_ccc_report_final_6-1_13455.pdf

[15] Build Security In, DHS Office of Cybersecurity and Communications, "Software Assurance", https://buildsecurityin.us-cert.gov/bsi/mission.html

[16] "15th Annual 2010/2011 Computer Crime and Security Survey", Computer Security Institute, http://scadahacker.com/library/Documents/Insider_Threats/CSI%20-%202010-2011%20Computer%20Crime%20and%20Security%20Survey.pdf

[17] IBM 2011 Annual Report, http://www.ibm.com/investor/pdf/2011_ibm_annual.pdf

[18] http://www.klocwork.com/blog/press-releases/klocwork-announces-record-quarterly-and-annual-results-for-2012/

[19] http://blogs.gartner.com/neil_macdonald/2012/01/30/interactive-application-security-testing/

[20] https://github.com/secdec/codepulse

[21] https://www.owasp.org/

7 ACRONYM LIST

API	Application Programming Interface
BCI	Byte-Code Instrumentation
CCS	Cascading Style Sheets
CPU	Central Processing Unit
CVEs	Common Vulnerabilities and Exposures
DAST	Dynamic Application Security Testing
DESA	Dynamic Enhanced Static Analysis
DHS	Department of Homeland Security
DT	Dynamic Tracing
FISMA	Federal Information Security Management Act
GPL	General Public License
GPU	Graphics Processing Unit
HIPAA	Health Insurance Portability and Accountability Act
HTML	HyperText Markup Language
IAST	Interactive Application Security Testing
IT	Information Technology
IDE	Integrated Development Environment
JAR	Java ARchive
JDI	Java Debug Interface
JDK	Java Development Toolkit
JSP	Java Server Pages
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiler Interface
OWASP	Open Web Application Security Project
PCI_DSS	Payment Card Industry Data Security Standard
RCP	Rich Client Platform
SAST	Static Application Security Testing
SWAMP	Software Assurance Market Place
TCP	Transmission Control Protocol
UI	User Interface
ZAP	Zed Attack Proxy